October 1992

UILU-ENG-92-2240
CRHC-92-22

Center for Reliable and High-Performance Computing

MEASUREMENT AND ANALYSIS OF OPERATING SYSTEM FAULT TOLERANCE

I. Lee, D. Tang, and R. K. Iyer

Coordinated Science Laboratory
College of Engineering
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

Measurement and Analysis of Operating System Fault Tolerance

Inhwan Lee, Dong Tang, and Ravishankar K. Iyer

Center for Reliable and High-Performance Computing
Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
1101 West Springfield Avenue
Urbana, IL 61801

September 30, 1992

Abstract

This paper demonstrates a methodology to model and evaluate the fault tolerance characteristics of operational software. The methodology is illustrated through case studies on three different operating systems: the Tandem GUARDIAN fault-tolerant system, the VAX/VMS distributed system, and the IBM/MVS system. Measurements are made on these systems for substantial periods to collect software error and recovery data. In addition to investigating basic dependability characteristics such as major software problems and error distributions, we develop two levels of models to describe error and recovery processes inside an operating system and on multiple instances of an operating system running in a distributed environment. Based on the models, reward analysis is conducted to evaluate the loss of service due to software errors and the effect of the fault-tolerance techniques implemented in the systems. Software error correlation in multicomputer systems is also investigated.

Results show that I/O management and program flow control are the major sources of software problems in the measured IBM/MVS and VAX/VMS operating systems, while memory management is the major source of software problems in the Tandem/GUARDIAN operating system. Software errors tend to occur in bursts on both IBM and VAX machines. This phenomenon is less pronounced in the Tandem system, which can be attributed to its fault-tolerant design. The fault tolerance in the Tandem system reduces the loss of service due to software failures by an order of magnitude. Although the measured Tandem system is an experimental system working under accelerated stresses, the loss of service due to software problems is much smaller than that in the measured VAX/VMS and IBM/MVS systems. It is shown that the software Time To Error distributions obtained from data are not simple exponentials. This is in contrast with the common assumption of exponential failure times made in fault-tolerant software models. Investigation of error correlations show that about 10% of software failures in the VAXcluster and 20% in the Tandem system occurred concurrently on multiple machines. The network-related software in the VAXcluster and the memory management software in the Tandem system are suspected to be software reliability bottlenecks for concurrent failures.

I. Introduction

While hardware reliability has improved significantly in the past decades, improvements in software reliability have not been quite as pronounced. Hardware faults are generally well-understood and efficient hardware fault models exist; hardware fault tolerance is relatively inexpensive to implement. Unfortunately none of these is true for software. It is generally believed that software is the major source of system outages in fault-tolerant systems [Gray90]. Software faults are logically complex, poorly-understood, and hard to model. Besides, software fault tolerance is a moving target because software quality keeps changing with design updates and software varies significantly from system to system. To complicate matters, software interacts with hardware and the environment, blurring the boundaries between them.

This paper presents a common methodology to model and evaluate the fault tolerance characteristics of operational software. The methodology is illustrated through case studies on three different operating systems: the Tandem/GUARDIAN fault-tolerant system, the VAX/VMS distributed system, and the IBM/MVS system. A study of these three operating systems is interesting because they are widely used and are representative of the diversity in the field. The Tandem/GUARDIAN and VAX/VMS data provide high-level information on software fault tolerance. The MVS data provides detailed information on low-level error recovery. Our intuitive observation is that GUARDIAN and MVS have a variety of software fault tolerance features, while VMS has less explicit software fault tolerance.

Ideally, we would like to have measurements on different systems under identical conditions. The reality, however, is that differences in operating system architectures, instrumentation conditions, measurement periods, and operational environments make this ideal practically impossible. Hence, a direct and detailed comparison between the systems is inappropriate. However, it is worthwhile to demonstrate the applications of a modeling and evaluation methodology on different systems. Also, these are mature operating systems which are slow-changing and have considerable common functionality. Thus, the major results can provide some high-level comparisons that point to the type and nature of relevant dependability issues.

The methodology proposed consists of the following steps:

(1) Error data reduction and classification — This step identifies software errors/failures and categorizes them by the affected system functions.

(2) Characterization of error time distributions — This step obtains empirical distributions to describe software error detection and recovery characteristics.

(3) Modeling and reward analysis — This step performs two levels of modeling and reward analysis.

(a) Low-level modeling of error detection and recovery in an operating system: this is illustrated using the IBM/MVS data.

(b) High-level modeling and evaluation of loss of work in a distributed environment: this is illustrated using the Tandem/GUARDIAN and VAX/VMS data.

(4) Investigation of error correlations — This step identifies concurrent software errors/failures on multiple instances of an operating system in a distributed environment and their sources.

The next section discusses the related research. Sections 3 and 4 describe measurements and error classifications. Section 5 characterizes error time distributions. Section 6 builds two levels of models to describe software fault tolerance and performs reward analysis to evaluate software dependability. Section 7 investigates errors correlations in distributed environments. Section 8 summarizes this study.

II. Related Research

Software reliability modeling has been studied extensively and a large number of models have been proposed [Musa87]. However, modeling and evaluation of fault tolerant software systems are not well understood. A discussion of key issues appeared in [Hecht86]. An analysis of failures and recovery of the MVS operating system running on an IBM 3081 machine was given in [Velardi84]. The study showed that 25% to 35% of all software failures were hardware-related and the system failure probability for hardware-related software errors was close to three times that for all software errors in general [Iyer85a]. A detailed analysis of software error and recovery in the MVS operating system was discussed in [Hsueh87]. More recently a wide-ranging analysis of failures in the MVS operating system and IBM database management systems is reported in [Sullivan91, Chillarege92].

Analytical modeling of fault tolerant software has been provided by several authors. In [Laprie84], an approximate model was derived to account for failures due to design faults; the model was used to evaluate a fault-tolerant software system. In [Scott87], several reliability models were used to evaluate three different software fault tolerance methods. Recently, more detailed dependability modeling and evaluation of two major software fault tolerance approaches — recovery blocks and N-version programming — have been proposed [Arlat90].

Although an operating system is a complex software system, little work has been done on modeling and evaluation of fault tolerance on operating systems. Major approaches for software fault tolerance rely on design diversity [Avizienis84, Randell75]. However, these approaches are usually inapplicable to large operating systems due to immense cost in developing and maintaining these software. Still each of the measured operating systems has distinct fault tolerance features. Single-failure tolerance of the Tandem system — although not explicitly intended for tolerating software design faults — provides the GUARDIAN system with a significant level of fault tolerance. In the MVS system, software fault tolerance is provided by *recovery management*. The philosophy in MVS is that the programmer who writes a critical system function envisages typical failure scenarios and provides recovery routines for each failure, so as to prevent a total system loss.

III. Measurements

For this study, measurements were made on three different operating systems: the Tandem/GUARDIAN system, the VAX/VMS system, and the IBM/MVS system. Table 3.1 summarizes the measured systems. These systems are representative of the diversity in the field in that they have varying degrees of fault tolerance embedded in the operating system. The Tandem/GUARDIAN system provides recovery from a wide range of software errors via the "Nonstop" environment. The IBM/MVS system includes robust recovery management to tolerate software-related errors. The Quorum algorithm (to be discussed later) implemented in the distributed VAX/VMS system running on a VAXcluster makes the VAXcluster function as a k-out-of-n fault-tolerant system. The following subsections introduce the three systems and measurements.

Table 3.1. Summary of Systems

HW System	SW System	Architecture	Fault-Tolerance	Workload
Tandem Cyclone	GUARDIAN	Distributed	Single-Failure Tolerance	SW Development/Testing
VAXcluster	VMS	Distributed	Quorum Algorithm	Scientific Applications
IBM 3081	MVS	Single	Recovery Management	System Design/Development

A. Tandem/GUARDIAN

The Tandem GUARDIAN system is a loosely-coupled multiprocessor system built for on-line transaction processing [Katzman78]. High availability is achieved via single-failure tolerance. With multiple processors running

process pairs, dual interprocessor buses, dual-port device controllers, disk mirroring, and redundant power supplies, a single failure in a processor, bus, device controller, disk, or power supply can be tolerated. Key software components in the Tandem system are processes and messages [Bartlett78]. From a software perspective, a Tandem system can be viewed as a collection of processes constantly sending and receiving messages. The GUARDIAN operating system has extensive low-level software error detection mechanisms. GUARDIAN is also capable of detecting errors using the "I'm alive" message protocol. Each processor periodically sends an "I'm alive" message to all processors on the system including itself. If the operating system in a processor does not receive the "I'm alive" message from another processor, it declares that processor to be failed. Detected errors are corrected by software either at component or system level using redundancy.

The data for this study was obtained from the processor halt log. The processor halt log is a subset of the TMDS (Tandem Maintenance and Diagnostic System) event log maintained by the GUARDIAN operating system. This log consists of events generated by three types of sources: the operating system, hardware, and human intervention. A *software halt* occurs when the operating system in a processor detects a problem that cannot be resolved. According to experienced Tandem engineers, software halts are mostly related to real software problems (i.e., software bugs). This is plausible considering the extensive hardware-error detection mechanisms in the Tandem system. Once a software halt occurs in a processor, a memory dump is taken from the processor, and the processor is reloaded. A fix is made later on based on the results of diagnosis using the memory dump.

Measurements were made on five systems — one field system and four in-house systems — for over a total of five system-years. Software halts are rare in the Tandem system and only one of the in-house systems had enough software halts for a meaningful analysis. This system was a Tandem Cyclone system used by Tandem software developers for a wide range of design and development experiments. It was operating as a beta site and was configured with old hardware. Sometimes the system was deliberately faulted for analysis. As such it is not representative of the Tandem system in the field. The measured period was 19 months (from July 1990 to January 1992).

B. VAX/VMS

The VAX/VMS error data was collected from two DEC VAXclusters. A VAXcluster is a distributed computer system consisting of several VAX machines and mass storage controllers. These machines and controllers (nodes) are connected by the Computer Interconnect (CI) bus organized as a star topology. Features of the VAXcluster include: nodes communicating based on a message-oriented interconnect and memory-to-memory block transfers, sharing of disk storage through the network (CI bus and ports), and running a distributed version of the VAX/VMS operating system [Kronenberg86].

The VAX/VMS distributed operating system provides sharing of resources (devices, files, and records) among users. Major components to provide resource sharing include the *file and record management services, disk class driver, lock manager, connection manager,* and *SCA software* [Kronenberg86]. These routines manage the cluster-wide resource sharing and communications. They are also responsible for coordinating the cluster members and handling recoverable failures in remote nodes. For the purpose of this study, we will call all these software components *I/O management routines.*

One of the VAXcluster design goals is to achieve high-availability by integrating multiple machines in a single system. The Quorum algorithm [Kronenberg86] implemented in the distributed VAX/VMS system makes the VAXcluster function as a k-out-of-n system. Each operating system running in the VAXcluster has a parameter called VOTES and a parameter called QUORUM. If there are n machines in the system, each operating system usually sets its QUORUM to $\lfloor n/2+1 \rfloor$. The parameter VOTES is dynamically set to the number of machines currently alive in the VAXcluster. The processing of the VAXcluster proceeds only if VOTES is greater than or equal to QUORUM. Thus, the VAXcluster functions like an $\lfloor n/2+1 \rfloor$-out-of-n system.

The first system, VAX1, consisted of seven machines and four controllers. The data collection periods for the different machines in VAX1 varied from 8 to 10 months (during October 1987 — August 1988). The cumulative measurement time was 5.5 machine years. The second system, VAX2, consisted of four machines and one controller. The data collection period was 27 months (January 1989 — March 1991). There were also 16 other small machines connected to VAX2 through an Ethernet during the measured period. Measurements were also made on these machines. The cumulative measurement time for VAX2 was 25.7 machine years. The two systems were used to provide service for scientists, engineers, and students for their research during the measured periods. The source

of the error data was the ERROR_LOG files produced by the VAX/VMS operating system.

C. IBM/MVS

The MVS is a widely used IBM operating system. Primary features of the system are reported to be efficient storage management and automatic software error recovery. In the MVS environment errors are detected by both hardware and software facilities. The hardware detects conditions such as memory violations, program errors (e.g., arithmetic exceptions), and addressing errors. The software detects more complex error conditions. The *data management and supervisor routines* ensure that valid data are processed and non-conflicting requests are made. For example, these routines can detect an incorrect parameter specification in a system macro, or a supervisor call issued by an unauthorized program.

The MVS system attempts to correct software errors using recovery routines. The philosophy in the MVS is that for each major system function, the programmer envisages possible failure scenarios and writes a recovery routine for each. However, it is the responsibility of the installation (or the user) to write recovery routines for other programs. The installation can improve the error detection capability of the system by means of a software facility called Resource Access Control Facility (RACF). The RACF is used to build detailed profiles of system software modules. These profiles are used to inspect the correct usage of system resources. The user can also employ other software facilities to detect the occurrences of selected events. In addition, the operator can detect some evident error conditions and decide to cancel or restart a job.

The detection of an error is recorded by an operating system module. The software record contains the information about the event that caused the record to be generated and a 12-bit error symptom code describing the reason for the abnormal termination of a program. The total number of implemented error codes is more than 500.

Measurements were made on an IBM 3081 mainframe running the IBM/MVS operating system. The system consisted of dual processors with two multiplexed channel sets. Time-stamped, low-level error and recovery data on errors affecting the operating system functions were collected. During the measured period, the system was used primarily to provide a time-sharing environment to a group of engineering communities for their daily work on system design and development. The measurement period was one year.

IV. Error Classification

In this section we identify software errors by processing the raw data and categorize these errors by the affected system function. A common step in data processing for all these systems is the *data coalescing*. In a computer system, a single problem commonly result in many repeated error observations occurring in rapid succession. To ensure that the analysis is not biased by repeated observations of the same problem, all error entries which have the same error type and occur within a short time (usually five minutes) interval of each other should be coalesced in data processing. However, the coalescing algorithms used in the three systems are not identical. Also, the error classification is different for the three systems. This is because these systems have different hardware and software architectures and error detection mechanisms. In the following discussion, major sources of software problems for these systems are identified by statistical analysis. This information can only be obtained from measurement-based analysis and can be used in designing techniques for software fault tolerance.

A. Tandem/GUARDIAN

The first step of the analysis was to identify event clusters for individual operating systems using the software failure data from the processor halt log. An event cluster consists of a sequence of related events ending with an event representing the recovery of a failed operating system. An event cluster represents an operating system failure and identifies the time period during which the operating system is unavailable. We also created event clusters using the non-software failure data from the processor halt log in a similar fashion. Further details of the data processing are given in [Lee91].

The processor halt log also provides information about the instruction processing environments (i.e., apparent causes of software halts from the operating system perspective and processes which were executing) prior to the occurrence of software halts. Table 4.1 shows the apparent causes of the collected software halts. The codes A, B, C, and D in the third column of the table will be used to refer to the cause later. The table shows that software halts occurred most frequently due to problems in accessing the system disk. Each processor in the system has its own copy of the kernel in its main memory, but it relies on the system disk, which is accessed through either processor 0 or 1, for all additional operating system related procedures and files. As a result, a problem in accessing the system disk can cause software halts in multiple processors on the system. This explains why it was the most frequent

Table 4.1. Apparent Cause of Software Halt (Tandem)

Software Subsystem	# Events	Cause Breakdown	# Events
Process Control	32	(A) illegal address reference by interrupt handler or system process	9
		(B) arithmetic overflow occur to interrupt handler or system process	4
		(C) memory manager read error, potentially related to system disk access	18
		(D) program internal error	1
Memory Management	7	(A) page or byte unlock error	1
		(B) illegal attempt to deallocate segment or bad segment table entry	3
		(C) unexpected page fault or too many errors on a processor switch	2
		(D) physical-page table and segment page table mismatch	1
Message System	14	(A) processor unsynchronized	1
		(B) unable to send "I'm alive" message to itself	6
		(C) declared down by the other live processors	7
Processor Control	1	instruction failure or sequence error on boot	1
Hardware-Related	21	(A) uncorrectable memory error	1
		(B) software encounter unexpected problem during recovery from power failure	15
		(C) unexpected interrupt or trap	5
Application Software	4		
Unknown	3		

cause of the software halts. Problems with the system disk can be caused by software faults or double component failures such as near-coincident halts in processors 0 and 1.

The second most frequent cause of software halts was an unexpected problem such as insufficient status information found by software during a recovery from a power failure. This could be due to design faults in the software that handles such recovery. The third most frequent cause of software halts was an illegal address reference by an interrupt handler or a system process.[1] Illegal address references can occur due to underlying software faults or undetected hardware errors. Other frequent causes of software halts were failures in sending or receiving the "I'm alive" message. The underlying cause of this can be an operating system failure, a processor hardware failure, or interrupt bursts from faulty hardware. The interrupt bursts would interfere with the message interrupt handling. The "I'm alive" protocol allows problems to be identified by the operating system on which the problem occurs, or by

[1]System processes consist of a small number of privileged processes. Examples are the monitor process, the memory manager, the operator process, and the I/O processes.

other operating systems. Obviously this protocol plays an active role in detecting processor software halts at a high level.

Table 4.2 shows a breakdown of the software halts based on the processes which were executing prior to the occurrence of software halts. The process that was executing points to the likely problem source. The third column of the table lists the cause of the halt (Table 4.1 nomenclature). "Non-process" refers to no specific process and indicates that an interrupt handler or special operating system procedure such as an idle loop was executing. The system monitor runs in each processor on the system. This process handles housekeeping functions and initiates process creation and deletion within a processor. Like the system monitor, the memory manager runs in each processor. This process services special requests from page-fault interrupt handler to bring needed pages into processor memory from disk. "All others" refers to a collection of about twenty processes. No more than two software halts occurred while each of these processes was executing.

The table clearly shows that software halts occurred mostly frequently while the memory manager process was executing to provide services to user applications. The most frequent cause of these halts was problems in accessing the system disk. The second most frequent cause was illegal address references by the memory manager process. The occurrence of software halts due to this cause did not show specific pattern. The fact that there were eight software halts due to this cause (Tables 4.1 and 4.2) indicates that these are likely to be due to underlying software faults.

B. VAX/VMS

Table 4.2. Active Process at Software Halt (Tandem)

Process	# Events	Cause Breakdown	# Events
Non-process	22	memory management (C)	2
		message system (A, B, C)	6
		hardware-related (A, B)	13
		unknown	1
System Monitor	4	processor control	1
		memory management (B)	3
Memory Manager	28	process control (A, C)	27
		memory management (D)	1
Unknown	4	hardware-related (C)	4
All Others	24		

The time-stamped VAX log contained information on both errors and failures. An *error* was defined as an abnormality in any component of the system. If an error led to a termination of the system service on a machine, it was defined as a *failure*. A failure was identified by a reboot report following one or multiple error reports. Based on the information provided by the error logs, errors were classified into three categories: *hardware*, *software*, and *unknown*. Hardware errors included five different types: *CPU*, *memory*, *disk*, *tape*, and *network* errors. Errors of the same type occurring within five minutes of each other were coalesced into a single *error event*. Details of the measurements and data processing can be found in [Tang92a].

Software errors were identified from "bugcheck" reports in the log files and divided into three types:

(1) Control — Problems involving program flow control or synchronization. For example, "Unexpected system service exception", "Exception while above ASTDEL or on interrupt stack", and "Spinlock(s) of higher rank already owned by CPU".

(2) Memory — Problems related to memory management or usage. For example, "Bad memory deallocation request size or address", "Double deallocation of memory block", "Page fault with IPL too high", and "Kernel stack not valid".

(3) I/O — Inconsistent conditions detected by I/O management routines. For example, "Inconsistent I/O data base", "RMS has detected an invalid condition", "Fatal error detected by VAX port driver", "Invalid lock id", and "Insufficient nonpaged pool to remaster locks on this system".

Table 4.3 shows the software error frequency, percentage, and associated failure probability, by error type for the measured VAX/VMS systems. The failure probability is defined as the failure frequency divided by the error frequency. It is seen that control errors are the dominant type of error (57%). However, closer examination of the data shows that some of control errors were related to network errors, which probably occurred in I/O management routines. These errors should be regarded as I/O errors. That is, the actual percentage of I/O errors is higher than that (36%) in the table. Thus, for the measured VAX/VMS systems, major software problems are from program flow control and I/O management.

The table shows that the failure probability of software errors is high (0.76). This is in contrast with the low failure probability of hardware errors (<0.01) shown in [Tang92a]. In the two VAXclusters, software failures

constitute approximately 23% of all machine failures. That is, the impact of software failures on system dependability for the measured systems is significant.

Table 4.3. Statistics for VAX/VMS Software Errors

Type	Control	Memory	I/O	All
Error Frequency	97	12	60	169
Percentage	57.4	7.1	35.5	100
Failure Probability	0.86	0.58	0.68	0.76

C. IBM/MVS

For the MVS system, two levels of data coalescing were performed [Hsueh87]. First, identical error entries which occurred within five minutes of each other were coalesced into a single record. Second, different error records occurring in close proximity in time (i.e., within 15 minutes of each other) were merged into a single error group. The first step eliminates multiple identical entries due to the same problem. The second step provides a mechanism to identify error bursts containing multiple but different error records. The reduced software errors were then classified into the following eight types:

(1) Control (CTRL) — invalid use of control statements or invalid supervisor calls

(2) Deadlocks (DLCK) — endless loops or wait states, or violation of system- or user-defined time limits

(3) I/O & Data Management (I/O) — errors occurring during I/O management or during creation/processing of data sets

(4) Storage Management (SM) — errors in storage allocation/deallocation or in virtual memory mapping

(5) Storage Exceptions (SE) — addressing of nonexistent or inaccessible memory locations

(6) Programming Exceptions (PE) — program errors (e.g., arithmetic overflow) other than storage exceptions

(7) Others (OTHR) — errors which don't fit any of the above categories

(8) Multiple Errors or Error Bursts (MULT) — error bursts consisting of different types (listed above) of errors

Table 4.4 lists the frequencies of the software errors defined above. The table shows that more than a half (52.5%) of the software errors were I/O & data management errors, i.e., the major source of software errors is I/O & data management. A significant percentage (17.4%) of the errors were classified as multiple errors.

Table 4.4. Statistics for IBM/MVS Software Errors

Type of Errors	Frequency	Percent
Control	213	7.72
Deadlock	23	0.84
I/O & Data Management	1448	52.50
Program Exceptions	65	2.43
Storage Exceptions	149	5.40
Storage Management	313	11.35
Others	66	2.32
Multiple Error	481	17.44
Total	2758	100.00

D. Summary

In the IBM/MVS system, the major source of software errors was I/O & data management. This is in agreement with the results from the VAX/VMS system where program flow control and I/O management were the major sources of software errors. However, memory management was the major source of software halts in the Tandem/GUARDIAN system. This difference is mainly attributed to the different hardware and software architectures among these systems. In the VAXcluster, programs highly rely on resource sharing through communications across the network. A lot of problems arose from the network related hardware and software which are suspected to be a reliability bottleneck. In the Tandem system, memory management implicitly involves lots of activities including inter-process communications through inter-processor buses as well as disk accesses because each processor controls a subset of the disks in the system and files are distributed to all disks. As a result, there is a more chance of software halt occurring due to memory management.

V. Time To Error Distributions

Actual Time To Error (TTE) distributions are essential in evaluating software dependability. Often, for simplicity or due to lack of information, TTE or Time Between Error (TBE), and Time To Recovery (TTR) are assumed to be exponentially distributed [Arlat90]. An early measurement-based study showed that the software Time Between Failures (TBF) in a VM/CMS system had a Weibull distribution [Iyer85b]. This section investigates TTE (or TBE) and TTR distributions in the measured systems.

A. TBE Distributions

Before presenting TBE distributions, we first explain how a TBE distribution is obtained from a multicomputer system such as the VAXcluster or Tandem system. We use the VAXcluster to illustrate the procedure. In a VAXcluster, all machine members are working in a similar environment and running the same version of the VMS operating system. If the VAXcluster is viewed as a single resource, then every software error on all machines can be sequentially ordered and a distribution can be constructed. In this way, the whole system is treated as a single entity in which multiple instances of an operating system are running concurrently. The constructed TBE distribution describes the software error characteristics for the whole system. We will call this distribution the *multicomputer software TBE distribution.*

Figures 5.1, 5.2, and 5.3 show the empirical TBE or TTE distributions fitted to analytic functions using SAS [SAS85] for the three measured systems. All these distributions failed to fit simple exponential functions. The fitting was tested using Kolmogorov-Smirnov test or Chi-square test at 0.05 significance level. The two-phase hyperexponential distribution provided satisfactory fits for the VAXcluster and Tandem multicomputer software TBE distributions. An attempt to fit the MVS TBE distribution to a phase-type exponential distribution led to a large number of stages. As a result, the following multi-stage gamma distribution was used:

$$f(t) = \sum_{i=1}^{n} a_i g(t; \alpha_i, s_i) \tag{5.1}$$

where $a_i \geq 0$, $\sum_{i=1}^{n} a_i = 1$, and

$$g(t; \alpha, s) = \begin{cases} 0 & t < s, \\ \dfrac{1}{\Gamma(\alpha)} (t-s)^{\alpha-1} e^{-(t-s)} & t \geq s\ . \end{cases} \tag{5.2}$$

A five-stage gamma distribution provided a satisfactory fit.

The results show that the multicomputer software TBE distribution (VAX/VMS and Tandem/GUARDIAN) can be modeled as a probabilistic combination of two exponential random variables, which shows that there are two dominant error modes. The higher error rate, λ_2, with occurrence probability α_2, captures both the error bursts (multiple errors occurring on the same operating system within a short period of time) and concurrent errors (multiple errors on different instances of an operating system which interact with each other) on these systems. The lower error rate, λ_1, with occurrence probability α_1, captures regular errors and provides interburst error rate.

Figure 5.1. TTE Distribution (MVS)

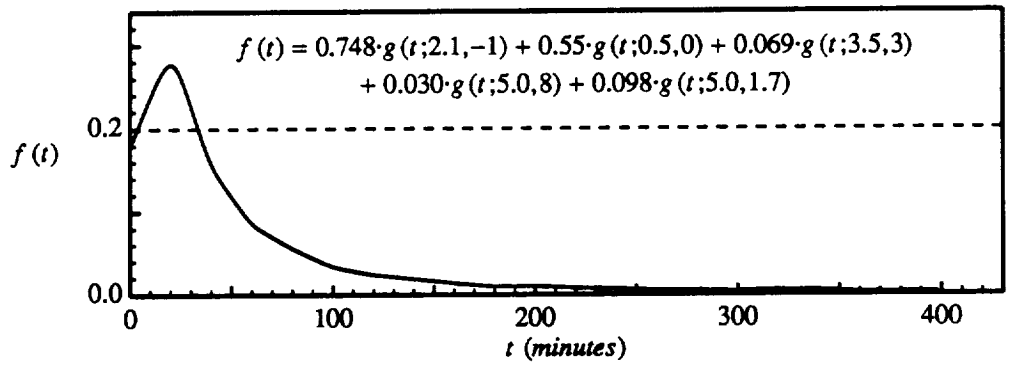


Figure 5.2. TBE Distribution (VAX1)

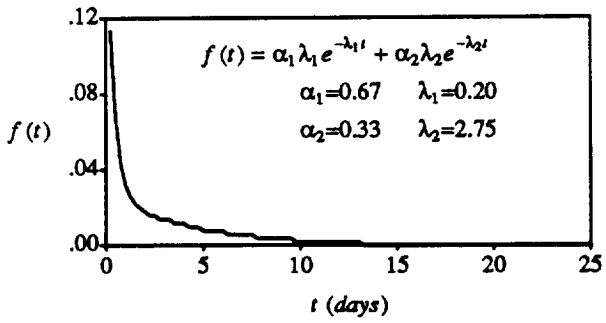
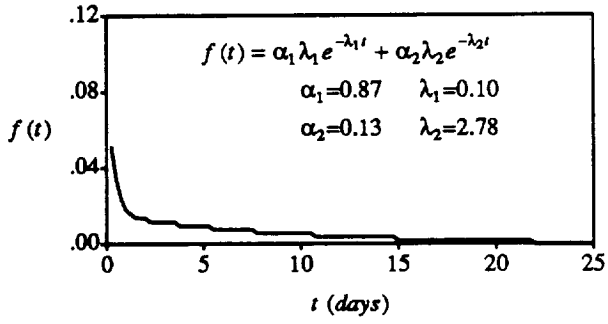


Figure 5.3. TBH Distribution (Tandem)



The VAX1 data (Figure 5.2) shows that a significant portion of the TBE instances (42%) are less than one day. This is in agreement with the existence of multiple errors (i.e., error bursts) in the IBM/MVS system. However, most of the VAXcluster TBE instances in the bursts are from a half hour to several hours which is longer than those instances in the IBM/MVS. A few TBE instances in VAX1 are less than a half hour. The software errors associated with these instances occurred on different machines. This indicates that some errors are correlated (this issue will be discussed further in section 7). Recall that we have used a coalescing algorithm to merge error reports within five minutes of each other. However, after this processing, there are still error burst phenomena.

These error bursts may be repeated occurrences of the same software problem, or multiple effects of an intermittent hardware fault on the software. Actually, software error bursts have been observed in laboratory experiments reported in [Bishop88]. The study showed that, if the input sequences of the software under investigation are correlated (rather than being independent), one can expect more "bunching" of failures than those predicted using a constant failure rate assumption. In an operating system, input sequences (user requests) are highly likely to be

correlated. Hence, a defect area can be triggered repeatedly.

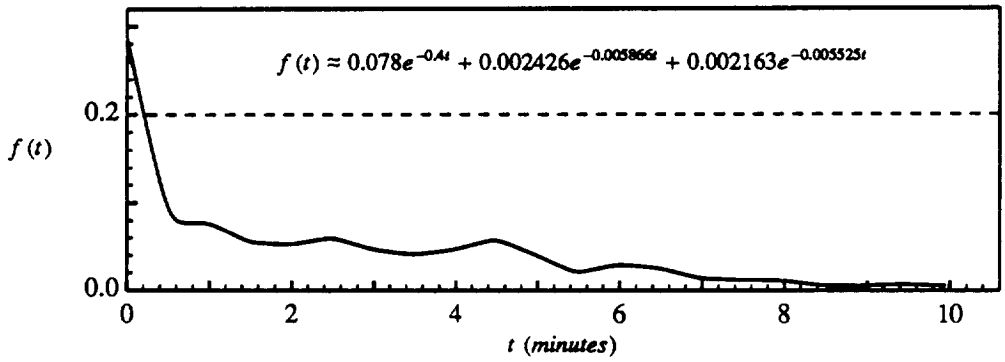
Figure 5.3 shows that the probability of error bursts in the Tandem system (0.13) is lower than that in the VAXcluster (0.32). That is, software halt bursts in the Tandem system are not as pronounced. This can be attributed to differences in the quality of software and error detection mechanisms. The Tandem system is designed for fault tolerance from the scratch, and the GUARDIAN operating system is believed to be more robust and less likely to cause repeated problems in a short time. In addition, the Tandem system is equipped with extensive hardware error detection mechanisms. Most hardware errors in the Tandem system are detected by hardware and corrected by software, and hence there is less chance that intermittent hardware errors cause repeated software errors.

It is clear that the measured TBE (or TTE) is not exponentially distributed. This result is in contrast with the typical assumption made in software dependability modeling [Arlat90]. The results do, however, conform with the previous measurements on IBM [Iyer85b] and DEC [Castillo81] machines. Several reasons for this non-exponential behavior, including the impact of workload, were documented in [Castillo81].

B. TTR Distributions

Figure 5.4 shows the spline-fit for the TTR distribution of multiple errors in the MVS system. The figure also shows an three-phase hyperexponential function to approximate the distribution. Figures 5.5 and 5.6 plot the empirical software TTR distributions for VAX1 and the Tandem system. Due to their peculiar shapes, we provided the raw distributions. Since most MVS software errors do not lead to system failures, the TTR for multiple errors,

Figure 5.4. MVS Multiple Error TTR Distribution (fitted)



which take the longest time to recover among all software error types in the MVS system, is still short. Most TTR instances are less than 10 minutes; suggesting that most often reload (at least temporarily) cleared the problems.

In VAX1 (Figure 5.5), most of the TTR instances (85%) are less than 15 minutes. This is attributed to those errors which are recovered by on-line recovery or automatic reboot without shutdown repair. However, some TTR instances last as long as several hours (the maximum is about 6.6 hours). These failures are, in our experience, probably due to a combination of software and hardware problems.

Figure 5.6 shows that the software recovery time in the Tandem system is longer than that in the VAXcluster. This is most likely due to the human factor in system maintenance. If a software halt occurs in the Tandem system, a memory dump is taken and then the halted processor is reloaded. Most software halts occur on one operating system. Since the system can tolerate a single operating system failure, operations tend to collect on-line information for diagnostic uses before attempting a recovery. This explains why the recovery time was longer in the Tandem system.

Typically, analytical models assume exponential or constant recovery times. Our results show that this does not apply generally. We have seen that all three TTR distributions are not simple exponentials. For the MVS system, since the recovery is usually quick, a constant recovery time assumption may be suitable. For the VAXcluster and Tandem systems, neither exponential nor constant recovery time can be assumed. More complex "multi-mode" functions may be needed to model these TTR distributions.

To summarize, significant software errors were found to occur in bursts on both IBM and VAX machines. This phenomenon is less pronounced in the Tandem system, which can be attributed to its fault-tolerant design. The

Figure 5.5. VAX1 Software TTR Distribution
(sample size = 95)

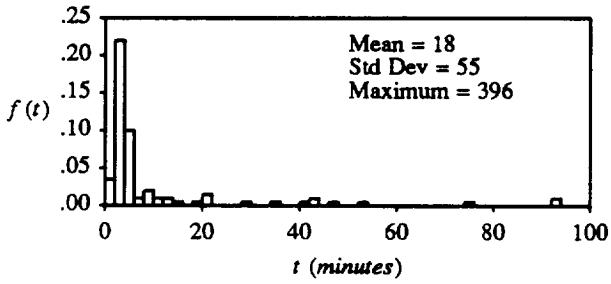
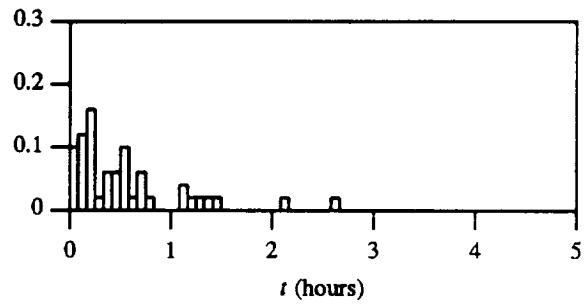


Figure 5.6. Tandem Software TTR Distribution
(sample size = 50)



measured software TBE and TTR distributions failed to fit simple exponential functions. This is in contrast with the common assumption of exponential failure times made in analytical modeling of software dependability. It was shown that a multicomputer software TBE distribution can be modeled by a two-phase hyperexponential random variable: a lower error rate captures regular errors, and a higher error rate captures error bursts and concurrent errors on multiple machines.

VI. Modeling and Analysis

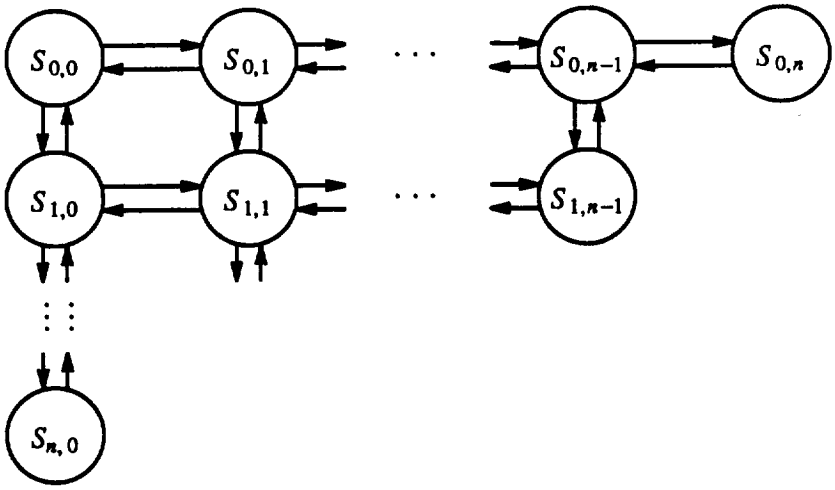
This section presents two levels of modeling and reward analysis to describe and evaluate the dynamic software dependability behavior. Low-level modeling focuses on error detection and recovery inside an operating system, while high-level modeling and reward analysis deals with distributed systems where multiple instances of operating systems interact. The IBM/MVS data is suited for illustrating the lower-level modeling, while the Tandem/GUARDIAN and VAX/VMS data are suited for illustrating the higher-level modeling and reward analysis. The two-level modeling and reward analysis not only allows us to evaluate software fault tolerance and software dependability, but also provides a framework to model complex software systems in a hierarchical fashion. We first presents the high-level modeling and reward analysis, and then presents the low-level modeling.

A. Distributed Operating System

In distributed environments such as the Tandem and VAXcluster systems, multiple instances of an operating system are running, and these instances form a single overall software system. This subsection discusses a high-level modeling to analyze this complex overall software system. Each instance of an operating system is treated as a software element of the overall software system, and software fault tolerance is discussed at a high level. The modeling is illustrated using the Tandem/GUARDIAN and VAX/VMS data.

The Tandem/GUARDIAN operating system closely interacts with hardware to provide software fault tolerance at system level. The GUARDIAN is robust and is equipped with extensive mechanisms to detect errors on software. Built-in single-failure tolerance allows the system to tolerate detected software errors using redundancy at system level. In a VAXcluster, the system-level software fault tolerance is provided by the Quorum algorithm. The VAXcluster can function as a $k-out-of-n$ system.

Figure 6.1. Model Structure



We constructed two-dimensional continuous-time Markov models using the software error data from the Tandem and VAXcluster systems. Figure 6.1 shows the model structure. In the model the state $S_{i,j}$ represents that, among a total of n systems, i are in an error state, and j are in a failure state, and $(n - i - j)$ are running error-free. The state transition probabilities were estimated from the measured data. For example, the state transition probability from state $S_{i,j}$ to state $S_{i,j+1}$ was obtained from

$$p_{(i,j),(i,j+1)} = \frac{\text{observed number of transitions from state } S_{i,j} \text{ to state } S_{i,j+1}}{\text{observed number of transitions from state } S_{i,j}} . \tag{6.1}$$

To quantify the impact of software errors on overall software dependability, we define two reward functions for the Markov models. The first applies to a non single-failure tolerant system such as the VAXcluster and the second applies to a single-failure tolerant system such as the Tandem system.

NSFT (No Single-Failure Tolerance) Reward Function:

We first define the reward function for a non single-failure tolerant system. In this case the system allows recovery from minor errors, but a major failure results in degradation. Given a time interval ΔT, a *reward rate* for a single operating system is defined as

$$r(\Delta T) = W(\Delta T) / \Delta T \tag{6.2}$$

where $W(\Delta T)$ denotes the useful service time provided by the operating system during the ΔT and is calculated by

$$W(\Delta T) = \begin{cases} \Delta T & \textit{in normal state} \\ \Delta T - c\tau & \textit{in error state} \\ 0 & \textit{in failure state} \end{cases} \tag{6.3}$$

where c is the number of raw errors occurring on the operating system during the ΔT and τ is the mean recovery time for a single error. Thus, one unit of reward is given for each unit of time when the operating system is in the normal state. In an error state, the reward loss depends on the amount of time the operating system spends on error recovery. (If ΔT is less than $c\tau$, $W(\Delta T)$ is set to 0.) In a failure state, the reward is zero. In a distributed environment, the reward is also inversely proportional to the number of failed systems.

With the above definition, the reward rate for state $S_{i,j}$ in the model (Figure 6.1) is obtained from

$$r_{i,j} = \frac{n - i\cdot\bar{c}\tau - j}{n} \tag{6.4}$$

where $\bar{c}$ is the average number errors occurring in an operating system, per unit time, in state $S_{i,j}$, and n is the total number of systems. Here each operating system failure causes degradation.

SFT (Single-Failure Tolerance) Reward Function:

The Tandem system allows recovery from minor errors and also a single major failure causes no loss of service in the Tandem system. To describe the built-in single-failure tolerance, we modify the reward rate (Equation 6.4) as follows:

$$r_{i,j} = \begin{cases} \dfrac{n - i\cdot\bar{c}\tau - j}{n} & \text{if } j = 0 \textit{ or } j = n \\ \dfrac{n - i\cdot\bar{c}\tau - j + 1}{n} & \text{if } 1 \le j \le (n-1)\,. \end{cases} \tag{6.5}$$

Here we assume that the system can tolerate a single operating system failure (i.e., a single processor software halt) without noticeable performance degradation. Thus the first failure causes no reward loss. For the second and subsequent failures, the reward loss is proportional to the number of these failures.

Given the Markov reward model described above, the expected steady-state reward rate, Y, can be estimated from [Trivedi92]

$$Y = \sum_{S_{i,j} \in S} r_{i,j} \cdot \Phi_{i,j} \tag{6.6}$$

where S is the set of valid states in the model and $\Phi_{i,j}$ is the steady-state occupancy probability for state $S_{i,j}$. The

steady-state reward rate represents the relative amount of useful service the system can provide per unit time in the long run, and can be regarded as a measure of software system service capability. The steady-state reward loss rate, $(1 - Y)$, represents the relative amount of service lost per unit time due to software errors in the long run. If we consider a specific group of errors in the analysis, the steady-state reward loss quantifies the performance loss due to this group of errors.

Table 6.1 shows the estimated steady-state reward loss for the Tandem and VAXcluster systems. The table shows the reward losses due to software as well as non-software problems. It is seen that software problems account for 27% of the service loss incurred by all problems in the Tandem system, while they account for 12% of the service loss incurred by all problems in the VAXcluster system. This indicates that software is not a dominant source of service loss in the measured VAXcluster system, while software is a significant source of service loss in the measured Tandem system. A census of Tandem system availability [Gray90] has shown that, as the reliability of hardware and maintenance improves significantly, software is the major source (62%) of outages in the Tandem system. Our results corroborate this finding.[2] Although the measured Tandem system was an experimental system, it has a smaller reward loss (by an order of magnitude) due to software problems. The same observation was made for the reward loss due to non-software problems. These observations demonstrate the high dependability of the measured Tandem system. In the VAXcluster, closer examination shows that most service loss due to non-software was actually incurred by hardware.

What Does Single-Failure Tolerance Buy?

Table 6.1. Steady-State Reward Loss

System	Measure	Software	Non-Software	Total
Tandem	Reward Loss	0.00006	0.00016	0.00022
	Percentage	27.3	72.7	100
VAXcluster	Reward Loss	0.00077	0.00565	0.00642
	Percentage	12.0	88.0	100

[2]Note that it is inappropriate to directly compare our number with Gray's because Gray's is an aggregate of many systems and ours is a measurement on a single system.

The Tandem/GUARDIAN data allows us to evaluate the impact of built-in software fault tolerance on system dependability and to relate loss of service to different software components. We performed reward analysis using the two reward rates defined above (SFT and NSFT) to evaluate the reduction in reward loss (loss of service) due to the software fault tolerance. The reward rate defined in Equation 6.4 allows us to determine the reward loss assuming no SFT. The reward rate defined in Equation 6.5 measures reward loss under SFT. The ratio of the two rewards estimated using these reward rate definitions determines the gain due to SFT. We evaluated the impact of seven different groups of halts on overall software dependability in the Tandem system: all software halts, the five mutually exclusive subsets of software halts (Table 4.4), and all non-software halts.

Table 6.2 shows the estimated steady-state reward loss due to the seven groups of halts for the two reward measures. The first row of the table shows that single-failure tolerance of the measured Tandem system reduced the loss of service incurred by all software halts by 96%, which clearly demonstrates the effectiveness of this fault tolerance mechanism against software failures. The table also shows that memory management is a potential dependability bottleneck in the Tandem system: the software halts which occurred while the memory manager process was executing account for 50 % of the loss of service incurred by all software halts (with SFT). The percentage was 25 with NSFT. A higher percentage of reward loss with SFT, compared with that with NSFT, indicates that the software halts which occurred while the memory manager process was executing tended to occur near-coincidentally on multiple operating systems.

The software halts under "all others" account for 65 % of the loss of service incurred by all software halts with NSFT. However, their real contribution to the measured system (with SFT) is about 17 %. This indicates that

Table 6.2. Steady-State Reward Loss (Tandem)

Source of Data	with SFT		with NSFT		$\frac{Reward\ Loss_{SFT}}{Reward\ Loss_{NSFT}}$
	Reward Loss	Percentage	Reward Loss	Percentage	
all s/w halts	0.00006	100 %	0.00136	100 %	4.4 %
s/w halts under non-process	0.00002	33 %	0.00012	9 %	16.7 %
s/w halts under system monitor	0.0	0 %	0.00000		0 %
s/w halts under memory manager	0.00003	50 %	0.00034	25 %	8.8 %
s/w halts under unknown	0.0	0 %	0.00000		0 %
s/w halts under all others	0.00001	17 %	0.00088	65 %	1.1 %
all non-s/w halts	0.00016		0.00206		7.8 %

these were mostly single operating system failures. The high steady-state reward loss with NSFT indicates that it took longer to recover from some of these halts. Interpretation of the impact of the software halts under "non-process" is less interesting because many of these were due to unexpected problems found by software during recoveries from power failures. This implies that the affected processors were already down due to power failures. Some of these halts occurred in two or three processors at almost the same time. There were not enough software halts under the system monitor or "unknown" to provide meaningful interpretations.

The last row of Table 6.2 shows that non-software halts caused more loss of service than software halts. This can be attributed to two reasons. First, some non-software halts which occur due to permanent hardware faults are resolved by replacing the faulty hardware, resulting in long recovery times and more loss of service. Second, non-software halts can occur due to environmental or operational faults. An environmental or operational fault can potentially affect all processors in the system. Both the first and last rows of Table 6.2 show that, although the measured Tandem system was in a high-stress environment, the steady-state reward loss is small (with SFT). This reflects the high dependability of the measured system. The last row also shows that single-failure tolerance reduced the loss of service incurred by all non-software halts by 92%. The percentage is slightly lower than that for software halts, which is attributed to the non-software halts, due to environmental or operational faults, which affected the whole system.

B. Single Operating System

This subsection discusses a low-level modeling to describe error detection and recovery inside an operating system. The methodology is illustrated using the IBM/MVS data. The MVS operating system provides software fault tolerance through recovery management. It also provides a flexible platform to build additional software fault tolerance at user level. Recovery routines in the MVS operating system provide a means by which the operating system prevents a total loss on the occurrence of software errors. When a program is abnormally interrupted due to an error, the supervisor routine gets the control. If the problem is such that further processing can degrade the system or destroy data, the supervisor routine gives the control to Recovery Termination Manager (RTM), an operating system module responsible for error and recovery management. If a recovery routine is available for the interrupted program, the RTM gives the control to this routine before it terminates the program. The purpose of a recovery

routine is to free the resources kept by the failing program, to locate the error, and to request either a retry or the termination of the program. Recovery routines are generally provided to cover critical MVS functions. However, it is the responsibility of the installation (or the user) to write recovery routines for other programs.

More than one recovery routine can be specified for the same program. If the current recovery routine is unable to restore a valid state, the RTM can give the control to another recovery routine, if available. This process is called *percolation*. The percolation process ends if either a routine issues a valid retry request or no more recovery routines are available. In the latter case, the program and its related subtasks are terminated. If a valid retry is requested, a retry is attempted to restore a valid state using the information supplied by the recovery routine and gives the control to the program. In order for a retry to be valid, there should be no risk of error recurrence and the retry address should be properly specified. An error recovery can result in the following four situations:

(1) Resume operation (Resume Op) — The system successfully recovered from the error and returned the control to the interrupted program.

(2) Task Termination (Task term)— The program and its related subtasks are terminated, but the system didn't fail.

(3) Job Termination (Job term)— The job in control at the time of the error is aborted.

(4) System Damage (System failure)— The job or task, which was terminated, is critical for system operation. As a result of the termination, a system failure occurs.

Using the collected error and recovery data, we constructed a continuous-time Markov model that provides a complete view of the measured MVS operating system from the error detection to the recovery. The states of the model consists of the eight different types of error states and the four states resulting from error recoveries. Figure 6.2 shows the model. The *normal* state represents that the operating system is running error-free. The transition probabilities were estimated from the measured data using Equation 6.1. Note that the system failure state is not shown in the figure. This is because the occurrence of system failure was rare and the number of observed system failures was statistically insignificant.

Table 6.3 shows the waiting time characteristics of the normal and error states in the model.[3] The table shows

[3] The waiting time for state i is the time that the system spends in state i before making a transition. Some researchers have used the term sojourn time for this measure.

Figure 6.2. Software Error/Recovery Model

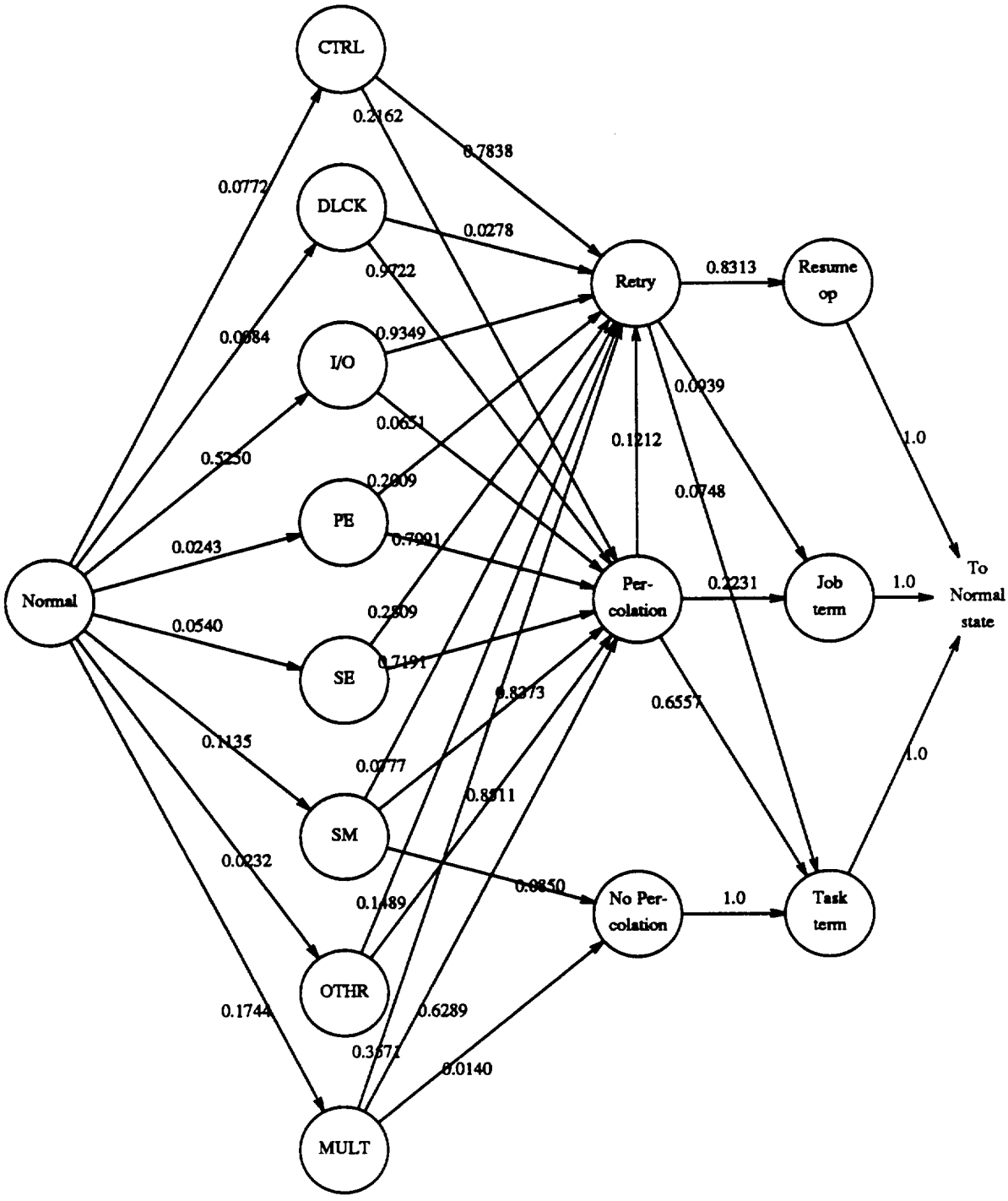


Table 6.3. Waiting Time

State	# of obs	Mean Waiting Time (in seconds)	Standard Deviation
Normal	2757	10461.33	32735.04
CTRL	213	21.92	84.21
DLCK	23	4.72	22.61
I/O	1448	25.05	77.62
PE	65	42.23	92.98
SE	149	36.82	79.59
SM	313	33.40	95.01
OTHR	66	1.86	12.98
MULT	481	175.59	252.79

that the duration of a single error is typically in the range of 20 to 40 seconds, except for deadlock (DLCK) and "others" (OTHR). The table also shows that errors of the type "others" (OTHR) are relatively insignificant because their average duration is less than 2 seconds. The average recovery time from a program exception is twice as long as that from a control error (21 seconds versus 42 seconds). This is probably due to the extensive software involvement in recovering from program exceptions. Table 6.3 clearly brings out the importance of incorporating multiple errors (or error bursts) into a system model. The average duration of a multiple error is at least four times longer than that of any types of single error.

An error recovery can be as simple as a retry or can be more complex requiring several percolations before a retry. The problem can also be such that no retry or percolation is possible. Figure 6.2 shows that about 83.1% of all retries are successful. The table also shows that the operating system attempts to recover from 93.5% of I/O and data management errors and 78.4% of control related errors by retries. These observations indicate that most I/O and control related errors are relatively easy to recover from, compared to the other types of errors such as deadlock and storage errors. Also note that "no percolation" occurs only in recovering from storage management errors. This indicates that storage management errors are more complicated than the other types of errors. The problem can also be such that no retry or percolation is possible.

Model Behavior:

The dynamic behavior of the modeled operating system can be described by various probabilities. Given the irreducible semi-Markov model of Figure 6.2, the following steady-state probabilities were evaluated. The derivations of these measures are given in [Howard71].

(1) transition probability (π_j) — given that the system is now making a transition, the probability that the transition is to state j

(2) occupancy probability (Φ_j) — at any instant of time the probability that the system occupies state j

(3) mean recurrence time ($\overline{\Theta}_j$) — mean recurrence time of state j

The occupancy probability of the normal state can be viewed as the operating system availability without degradation. The state transition probability, on the other hand, represents the dynamic behavior of the error detection and recovery processes in the operating system. Table 6.4(a) lists the state transition probabilities and occupancy probabilities for the normal and error states. Table 6.4(b) lists the state transition probabilities and the mean recurrent times of the recovery and result states. A dashed line in the table indicates a negligible value (less than 0.00001). Table 6.4(a) shows that the occupancy probability of the normal state in the model is 0.995. This indicates that in 99.5% of time the operating system is running error-free. In 0.5% of time the operating system is in error or recovery states. Table 6.4(b) shows that in more than half of this time (i.e., 0.29% out of 0.5%) the operating system is in the multiple error state. An early study on the MVS error/recovery showed that average reward rate for the software error/recovery state was 0.2736 [Hsueh88]. Based on this reward rate and the occupancy probability for the error/recovery state obtained above (0.005), we estimate that the steady-state reward loss is 0.00363 which is larger than that estimated for the Tandem and VAXcluster systems.

Table 6.4. Error/Recovery Model Characteristics

Measure	Normal state	Error state							
		CTRL	DLCK	I/O	PE	SE	SM	OTHR	MULT
π	0.2474	0.0191	0.0020	0.1299	0.0060	0.0134	0.0281	0.0057	0.0431
Φ	0.9950	0.00016	-	0.00125	0.000098	0.000189	0.00036	-	0.002913

(a)

Measure	Recovery state			Result		
	Retry	Percolation	No-Percolation	Resume op	Task term	Job term
π	0.1704	0.0845	0.0030	0.1414	0.0712	0.0348
$\overline{\Theta}^*$	4.25	8.55	241.43	5.11	10.16	20.74

* - in hour

(b)

By solving the model (Figure 6.2), it is found that the operating system makes a transition every 43.37 minutes. Table 6.4 shows that 24.74% of all transitions made in the model are to the normal state, 24.73% of them are to error states (obtained by summing all the π's for all error states), 25.79% of them are to recovery states, and 24.74% of them are to result states. Since a transition occurs every 43 minutes, we estimate that, on average, a software error is detected every three hours and a successful recovery (i.e., reaching the "resume op" state) occurs every five hours. This indicates that nearly 43% of all software errors result in task/job termination. Although very few (statistically insignificant number) of the task or job terminations lead to system failures, they do affect the user perceived reliability and availability of the system because the operating system recovers from more than 40% of the software errors via job or task termination. As a result of the termination, users may have to re-initiate the jobs/tasks. For long duration jobs/tasks, the performance loss (i.e., the loss of useful work) can be very high.

C. Summary

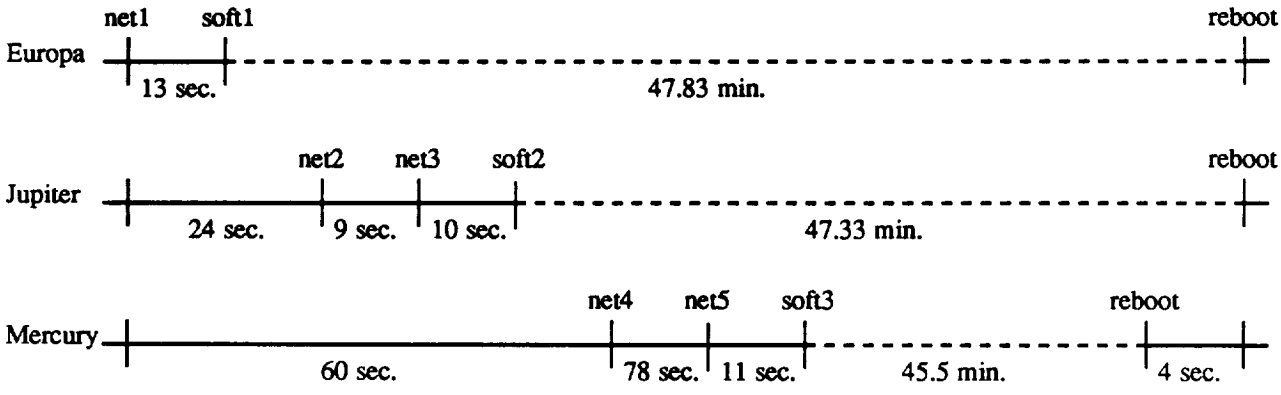
We presented two levels of modeling and reward analysis to describe and evaluate the dynamic software dependability behavior. The high-level modeling and reward analysis deals with distributed systems where multiple instances of operating systems interact, and was illustrated using the Tandem/GUARDIAN and VAX/VMS data. The low-level modeling focuses on error detection and recovery inside an operating system, and was illustrated using the IBM/MVS data. The results show that software is a significant source of service loss in the Tandem system, while hardware is the dominant source of service loss in the VAXcluster. Although the measured Tandem system is an experimental system working under accelerated stresses, the loss of service due to software problems is much smaller than that in the measured IBM/MVS and VAX/VMS systems. Further evaluation of the Tandem system shows that single failure tolerance of the Tandem system reduced the service loss due to software failures by an order of magnitude and the memory management software is a potential reward bottleneck. The analysis of the MVS data shows that, although few software errors caused system failures, user-perceived reliability and availability is low because more than 40% of all software errors result in user job/task termination. The MVS data also shows that multiple errors, which constitute over 17% of all software errors, have long recovery times and hence can impose a considerable system performance degradation.

VII. Error Correlation

When multiple instances of an operating systems interact with each other on a system, the issue of correlated errors/failures should be addressed. An early study shows [Tang92b] that even a small failure correlation or a few correlated failures can have a significant impact on system dependability. This section discusses software error correlations in the VAXcluster and Tandem systems.

We found that about 10% of software failures occurred on multiple machines concurrently in VAX1. It is instructive to examine these cases in detail to understand how software failures occurred concurrently on different machines. Figure 7.1 depicts a case scenario. In the figure, Europa, Jupiter, and Mercury are machine names in VAX1. A dashed line represents that the corresponding machine is in a failure state. At one time, a network error (net1) was reported from the CI (Computer Interconnect) port on Europa. This resulted in a software failure (soft1) 13 seconds later. Twenty-four seconds after the first network error (net1), additional network errors (net2,net3) were reported on the second machine (Jupiter), which was followed by a software failure (soft2). The error sequence on Jupiter was repeated (net4,net5,soft3) on the third machine (Mercury). The three machines experienced software failures concurrently for 45.5 minutes. All three software failures occurred shortly after network errors occurred. Thus, sometimes network-related hardware or software problems cause concurrent software failures on multiple machines.

Figure 7.1. A Scenario of Concurrent Software Failures



Note: soft1, soft2, soft3 — Exception while above ASTDEL or on interrupt stack.
net1, net3, net5 — Port will be re-started. net2, net4 — Virtual circuit timeout.

The VAXcluster data shows another danger of network-related errors in the form of *hardware-related software errors*. If a software error (failure) occurs in close proximity (e.g., within a minute) to a hardware error, it is referred to as a hardware-related software error (failure) [Iyer85a]. The occurrence of hardware-related software errors can be explained in several ways. First, hardware errors can be detected by software. For instance, a hardware error, such as a flipped memory bit, may change the software condition, causing a software error. This software error can either be corrected by built-in software fault tolerance or result in a software failure. Second, hardware errors can cause software failures by exercising parts of the operating system such as recovery routines which are rarely used, thus activating dormant software faults.

Table 7.1 shows the total counts of hardware-related software errors and failures, and their percentages to all errors and failures (including those caused by hardware), respectively, in the two VAXcluster systems. Each operating system was treated as an independent entity. The table shows that, out of 32 hardware-related software errors, 28 resulted in software failures, which shows the severity of hardware-related software errors. Closer examination of the data shows that network errors are responsible for most hardware-related software errors (75%).

Network errors are not only the major source of hardware-related software failures, but also related to a significant portion of correlated machine failures which are not reported as software failures [Tang92b]. All these failures are believed to occur in or relate to the network interface software in I/O management routines. Thus, we speculate that the network-related software is a reliability bottleneck in the measured VAXcluster systems.

The Tandem GUARDIAN data shows that about 20% of the software halts occurred concurrently on multiple operating systems. A significant portion of these concurrent software failures are believed to be related to software development/testing efforts on the measured system. However, there were three occasions where software halts occurring while the memory manager process was executing, resulted in system coldloads - restarts of all instances of the operating system. In one case, a single software halt due to an illegal address reference by the memory

Table 7.1. Hardware-Related Software Errors (VAXclusters)

Measure	Frequency	Percentage
Error	32	18.9
Failure	28	21.4

manager process resulted in a system coldload. In the other two cases, software halts occurred near-coincidentally on multiple processors due to problems in accessing the system disk. Recall that each operating system on the Tandem system relies on the system disk, which is accessed through either processor 0 or 1, for all additional operating system related procedures and files. As a result, these two processors and the system disk are more important than others from a dependability perspective. We believe that the high-stress environment of the measured system provided us with the rare opportunity to observe multiple component failure situations, and revealed this potential dependability bottleneck, i.e., problems in accessing the system disk. This burden can be alleviated by duplicating the procedures and files on another disk controlled by another pair of processors.

The above observations indicate that a failure of the measured Tandem system is likely to occur due to a single severe software halt or due to concurrent software halts on multiple processors caused by common causes (not due to a random coincidence of multiple independent software halts). Elimination of the underlying causes of these severe software halts will improve the reliability of the overall software. However, it should be cautioned that this study was based on an experiment on a Tandem system under accelerated stress. Further investigation is necessary based on more experiments and experiments on field systems.

In this section, we investigated correlated software failures and hardware-related software errors in the measured VAXcluster and Tandem systems. Correlated failures can stress recovery and break the protection provided by "single-failure" tolerance. To identify the sources of correlated failures is necessary for improving dependability. Our results show that the network-related software of the VAX/VMS and the memory management of the Tandem/GUARDIAN are potential software bottlenecks in terms of correlated failures. Providing additional fault tolerance in software to resolve these bottlenecks will significantly improve the dependability of these systems. Modeling of correlated failures has not been well addressed in literature. Model specification for correlated failures can be complex and additional stiffness may arise in the model solution because correlated failures are infrequent events. The fact that about 10% of software failures on VAX1 and 20% of software failures on the Tandem system occurred concurrently on multiple machines suggests that correlated failures cannot be neglected.

VIII. Conclusion

In this paper, we demonstrated a methodology to model and evaluate the fault tolerance characteristics of operational software. The methodology was illustrated through case studies on three different operating systems: the Tandem/GUARDIAN system, the VAX/VMS system, and the IBM/MVS system. Measurements were made on these systems for substantial periods to collect software error and recovery data. Major software problems and error characteristics were identified by statistical analysis. A two level modeling approach was used to model and evaluate error and recovery processes inside an operating system and on multiple instances of an operating system running in a distributed environment. Based on the models, reward analysis is conducted to evaluate the loss of service due to software errors and the effect of fault-tolerance techniques implemented in the systems. Software error correlation in multicomputer systems is also investigated.

Results show that I/O management and program flow control are the major sources of software problems in the IBM/MVS and VAX/VMS operating systems, while memory management is the major source of software problems in the Tandem/GUARDIAN operating system. Software errors tend to occur in bursts on both IBM and VAX machines. This phenomenon is less pronounced in the Tandem system, which can be attributed to its fault-tolerant design. The fault-tolerance in the Tandem system reduced its loss of service due to software failures by an order of magnitude. Although the measured Tandem system is an experimental system working under accelerated stresses, the loss of service due to software problems is much smaller than that in the measured IBM/MVS and VAX/VMS systems.

It is shown that the software Time To Error distributions obtained from the data are not simple exponentials. This is in contrast with the common assumption of exponential failure times made in fault-tolerant software models. Both the VAXcluster and Tandem data showed that a multicomputer software Time Between Error distribution can be modeled by a 2-phase hyperexponential random variable: a lower rate error pattern which characterizes regular errors, and a higher rate error pattern which characterizes error bursts and concurrent errors on multiple machines. Investigation on error correlations found that about 10% of software failures in VAX1 and 20% in the Tandem system occurred currently on multiple machines. It was suspected that the network-related software in the VAXcluster and memory management software in the Tandem system are software reliability bottlenecks, in terms of concurrent failures.

It should be emphasized that the results of this study should not be interpreted as a direct comparison between the three measured operating systems, but rather an illustration of the proposed methodologies. The differences in operating system architectures, instrumentation conditions, measurement periods, and operational environments make a direct comparison impossible.

Acknowledgements

We thank Tandem Computers Incorporated, NASA AMES Research Center, and IBM Poughkeepsie for their assistance in collecting the data from the Tandem machines, VAXcluster system, and IBM machines, respectively. Thanks are also due to Abhay Mehta of Tandem for providing technical support for the study of the Tandem system, and Robert Dimpsey, Kumar Goswami, and Wei-Lun Kao for their useful comments on this manuscript. This research was supported in part by the Office of Naval Research under Grant N00014-91-J-1116 and in part by NASA under grant NAG-1-613. The content of this paper does not necessarily reflect the position or policy of the government and no endorsement should be inferred.

References

[Arlat90] L. Arlat, K. Kanoun, and J.C. Laprie, "Dependability Modeling and Evaluation of Software Fault-Tolerant Systems," *IEEE Transactions on Computers*, Vol. 39, No. 4, April 1990, pp. 504-513.

[Avizienis84] A. Avizienis and J.P.J. Kelly, "Fault Tolerance by Design Diversity: Concepts and Experiments," *IEEE Computer*, August 1984, pp. 67-80.

[Bartlett78] J.F. Bartlett, "A 'Nonstop' Operating System," *Proceedings of the International Hawaii Conference on System Science*, 1978, pp. 103-117.

[Bishop88] P.G. Bishop and F.D. Pullen, "PODS Revisited — A Study of Software Failure Behavior," *IEEE FTCS-18*, June 1988, pp. 2-8.

[Castillo81] X. Castillo, *A Compatible Hardware/Software Reliability Prediction Model*, Ph. D. Thesis, Carnegie-Mellon University, July 1981.

[Castillo82] X. Castillo and D.P. Siewiorek, "A Workload Dependent Software Reliability Prediction Model," *IEEE FTCS-12*, Santa Monica, California, June 1982, pp. 279-286.

[Chillarege92] R. Chillarege and M.S. Sullivan, "A Comparison of Software Defects in Database Management Systems and Operating Systems," to appear in *IEEE FTCS-22*, Boston, MA, June 1992.

[Howard71] R. A. Howard, *Dynamic Probabilistic Systems*, John Wiley & Sons, Inc., New York, 1971.

[Gray90] J. Gray, "A Census of Tandem System Availability Between 1985 and 1990," *IEEE Transactions on Reliability*, Vol. 39, No. 4, Oct. 1990, pp. 409-418.

[Hecht86] H. Hecht and M. Hecht, "Software Reliability in the System Context," *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 1, Jan. 1986, pp. 51-58.

[Hsueh87] M.C. Hsueh and R.K. Iyer, "A Measurement-Based Model of Software Reliability in a Production Environment," *Proceedings of the 11th Annual International Computer Software & Applications Conference*, Tokyo, Japan, October 1987, pp. 354-360.

[**Hsueh88**] M.C. Hsueh, R.K. Iyer, and K.S. Trivedi, "Performability Modeling Based on Real Data: A Case Study," *IEEE Transactions on Computers*, Vol. 37, No. 4, April 1988, pp. 478-484.

[**Iyer85a**] R.K. Iyer and P. Velardi, "Hardware-Related Software Errors: Measurement and Analysis," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 2, February 1985, pp. 223-231.

[**Iyer85b**] R.K. Iyer and D.J. Rossetti, "Effect of System Workload on Operating System Reliability: A Study on IBM 3081," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 12, December 1985, pp. 1438-1448.

[**Katzman78**] J.A. Katzman, "A Fault-Tolerant Computing System," *Proceedings of the International Hawaii Conference on System Science*, 1978, pp. 85-102.

[**Kronenberg86**] N.P. Kronenberg, H.M. Levy and W.D. Strecker, "VAXcluster: A Closely-Coupled Distributed System," *ACM Transactions on Computer Systems*, Vol. 4, No. 2, May 1986, pp. 130-146.

[**Laprie84**] J.C. Laprie, "Dependable Evaluation of Software Systems in Operation," *IEEE Transactions on Software Engineering*, Vol. SE-10, No. 6, Nov. 1984, pp. 701-714.

[**Lee91**] I. Lee, R.K. Iyer and D. Tang, "Error/Failure Analysis Using Event Logs from Fault Tolerant Systems," *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, June 1991.

[**Musa87**] J.D. Musa, A. Iannino, and K. Okumoto, *Software Reliability: Measurement, Prediction, Application*, McGraw-Hill Book Company, 1987.

[**Randell75**] B. Randell, "System Structure for Software Fault Tolerance," *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 2, June 1975.

[**SAS85**] SAS Institute Inc., *SAS User's Guide: Statistics*, Version 5 Edition, Cary, NC, 1985.

[**Scott87**] R.K. Scott, J.W. Gault, and D.F. Mcallister, "Fault-Tolerant Software Reliability Modeling," *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 5, May 1987, pp. 582-592.

[**Sullivan91**] M.S. Sullivan and R. Chillarege, "Software Defects and Their Impact on System Availability — A Study of Field Failures in Operating Systems," *IEEE FTCS-21*, Montreal, Canada, June 1991.

[**Tang92a**] D. Tang and R.K. Iyer, "Measurement and Modeling of a Multicomputer System," *IEEE Transactions on Computers*, to appear.

[**Tang92b**] D. Tang and R. K. Iyer, "Analysis and Modeling of Correlated Failures in Multicomputer Systems," *IEEE Transactions on Computers*, May 1992.

[**Trivedi92**] K.S. Trivedi, J.K. Muppala, S.P. Woolet, and B.R. Haverkort, "Composite Performance and Dependability Analysis," *Performance Evaluation*, Vol. 14, February 1992, pp.197-215.

[**Velardi84**] P. Velardi and R.K. Iyer, "A Study of Software Failures and Recovery in the MVS Operating System," *IEEE Transactions on Computers*, Vol. C-33, No. 6, June 1984, pp. 564-568.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION Unclassified	1b. RESTRICTIVE MARKINGS None
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) UILU-ENG-92-2240 CRHC 92-22	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION Coordinated Science Lab University of Illinois	6b. OFFICE SYMBOL (If applicable) N/A
6c. ADDRESS (City, State, and ZIP Code) 1308 W. Main St. Urbana, IL 61801	7a. NAME OF MONITORING ORGANIZATION ONR and NASA
	7b. ADDRESS (City, State, and ZIP Code) ONR: Arlington, Virginia 22217 NASA: Hampton, Virginia 23665
8a. NAME OF FUNDING/SPONSORING ORGANIZATION ONR and NASA	8b. OFFICE SYMBOL (If applicable)
8c. ADDRESS (City, State, and ZIP Code) ONR: Arlington, Virginia 22217 NASA: Hampton, Virginia 23665	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER ONR: N00014-91-J-1116 NASA: NAG-1-613
	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE (Include Security Classification)

Measurement and Analysis of Operating System Fault Tolerance

12. PERSONAL AUTHOR(S)

L. Lee, D. Tang, and R. K. Iyer

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical	FROM _____ TO _____	October 1992	35

16. SUPPLEMENTARY NOTATION

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Fault-Tolerance, Software Dependability Evaluation, Operating System, Distributed System, Error Measurement, Error Recovery, Markov Model, Reward Analysis, Correlation

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This paper demonstrates a methodology to model and evaluate the fault tolerance characteristics of operational software. The methodology is illustrated through case studies on three different operating systems: the Tandem GUARDIAN fault-tolerant system, the VAX/VMS distributed system, and the IBM/MVS system. Measurements are made on these systems for substantial periods to collect software error and recovery data. In addition to investigating basic dependability characteristics such as major software problems and error distributions, we develop two levels of models to describe error and recovery processes inside an operating system and on multiple instances of an operating system running in a distributed environment. Based on the models, reward analysis is conducted to evaluate the loss of service due to software errors and the effect of the fault-tolerance techniques implemented in the systems. Software error correlation in multicomputer systems is also investigated.

continued

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT. ☐ OTIC USERS	Unclassified
22a. NAME OF RESPONSIBLE INDIVIDUAL	22b. TELEPHONE (Include Area Code) / 22c. OFFICE SYMBOL

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

UNCLASSIFIED